Security Lab – Hacking-Lab Challenges Part 4

Introduction

In this lab, you'll first do several Hacking-Lab challenges in the context of access control vulnerabilities and server-side request forgery (SSRF). In addition, the final challenge deals with a combination of various vulnerabilities, including information disclosure and deprecated functions. The goal is to better understand and gain practical experience with such attacks. To access the challenges, select event *Challenges Part 4* in the Hacking-Lab – this shows you the six challenges that are part of this event. It's recommended to work through the challenges in the given order.

1 GlockenEmil 2.0 – DOR Direct Object Reference

1.1 Goal

Get access to the full credit card information of other customers by exploiting an access control vulnerability. Assume that you are *customer0* with password *compass0*.

1.2 Required solution for full points

- A brief description of the security problem.
- The URL you used to access the credit card information of another customer.
- The credit card number and the CVV number of this customer.

1.3 Hints

- To access your own credit cards, log in and click on *Credit Cards* at the top right.
- Assume that there are other customers that have very similar credit card numbers to your own.

2 A Cookie for a Hacker

2.1 Goal

Get access to the very secret notes that were posted by an administrator, by exploiting an access control vulnerability.

2.2 Required solution for full points

- A brief description of the security problem.
- A brief description of how you managed to exploit the vulnerability.
- The flag that is included in the secret notes of the admin.

2.3 Hints

- To create notes, simply register using any username/password and post notes on the next web page you get.
- To do the attack, it may be helpful to use the *Intruder* component of *Burp* (which is rate-limited and slow in the available Community version, but it still can be used well to solve this challenge). Alternatively, you can use the *Fuzz* functionality of *OWASP ZAP* (which is very fast).

3 GlockenEmil 2.0 – JWT

3.1 Goal

Buy products at a reduced price by exploiting a vulnerability related to the handling of JSON Web Tokens (JWT) in the *GlockenEmil 2.0* shop.

JWTs are often used as authentication tokens in web applications that use a REST API in the backend. They are typically included in a request header as follows: *Authorization: Bearer eyJhbGci...* The API checks the JWT and, if everything is correct, grants access to the requested resource (e.g., based on the username or role included in the JWT). More details about JWTs will follow later in the module SWS1, but to solve this challenge, you don't have to know about the full details.

3.2 Required solution for full points

- A description of the attack you used, including the request you sent to exploit the vulnerability.
- A screenshot of the *My Orders* page of the shop that shows that you managed to reduce the price of an order.
- A brief description of the fundamental security problem that you managed to exploit.

3.3 Hints

- Log in, e.g., as *customer0*/*compass0* and click *Orders* at the top right to view your orders. Here, you can see several orders that have been made. The goal is to reduce the price of one of the orders so that in the end, the reduced price is shown.
- Use the *Web Developer Tools* of Firefox to view the network traffic and do a full reload of the page. Look for REST API calls and inspect the request headers. They should contain a JWT. You can view the content of the JWT by using *https://jwt.io*.
- Sometimes, REST APIs provide information about API usage. This is also the case here. Try different «reasonable» API URLs until you get information about available API methods (you'll only get information about one method). The information tells you how retailers can apply a discount to an order. Try to abuse this information to get a discount on one of the orders.

4 SCHOGGI: API Excessive Data Exposure

4.1 Goal

Get the password of at least one user by exploiting a so-called excessive data exposure vulnerability of the REST API of the Chocoshop. This means that one or more API endpoints return more data than necessary, or that API endpoints that shouldn't be available are actually available.

4.2 Required solution for full points

- A description of the attack steps you performed to get the password of at least one user.
- A screenshot that shows the usage of *john* and its output (which should include at least one cracked password). You can also use another tool for password cracking, if you like.

4.3 Hints

- Log in with *alice*/*alice.123*, click the links at the top, and analyze the API calls that deliver information about users.
- One of the REST principles says that meaningful URLs should be used. So, for instance, if */api/products* accesses all products, then something like */api/products/1234* or */api/product/1234/* may be used to access information about the product with ID *1234*.
- Once you managed to get all usernames and password hashes, you should try to crack the passwords. In the following, it is assumed that *john* is used for this. To use *john*, store the usernames and password hashes line-by-line in a file as follows:
  alice:<password hash of alice>
  bob::<password hash of bob>
  …

- *john* requires a wordlist as a basis to crack the passwords. A list of «the top 10'000 passwords» can be downloaded from:
  https://raw.githubusercontent.com/danielmiessler/SecLists/master/Passwords/Common-Credentials/10-million-password-list-top-10000.txt
- Assume the password hash is direct hash of the password without salt or multiple hashing rounds. Looking at the password hash should allow you to make a guess about the hash algorithm that is used (assume a frequently used hashing algorithm is used). The hash algorithms supported by john can be found, e.g., at *https://pentestmonkey.net/cheat-sheet/john-the-ripper-hash-formats*.
- Assuming the password hashes are stored in *hashes.txt*, the used hash algorithm is <*hash-algorithm*>, and the passwords in <*password-candidates-file*> should be tried, then *john* can be used as follows to crack the passwords:
  john hashes.txt --format=<hash-algorithm> --wordlist=<password-candidate-files>

5 GlockenEmil 2.0 - SSRF Server Side Request Forgery

5.1 Goal

Get access to an image with secret information by exploiting a Server-Side Request Forgery (SSRF) vulnerability. SSRF is an attack where a server makes a request on behalf of the attacker to access a resource (e.g., on the internal network or on the web server itself) that isn't directly accessible by the attacker. SSRF was already used in a previous challenge where an XML External Entity injection vulnerability was exploited. Here, a different vulnerability is involved.

5.2 Required solution for full points

- A description of the attack steps you performed to get the secret information.
- One of the passwords from the secret information.

5.3 Hints

- The main page shows some posts by customers of the shop. One of these posts doesn't contain an image, but just a link. Inspect this link and you'll see it is broken, but at the same time, this link should provide you with information about an internal resource. You can try to access this resource directly, but it won't work.
- Log in, e.g., as *customer0*/*compass0* and click *Community* at the top right. Here, you can create posts yourself. Try to abuse this to get access to the internal resource you identified before.
- Once you managed to do this, try to access another internal resource that contains secret information.

6 Historia Animalum

6.1 Goal

Get access to a secret flag hidden in the web application, by combining various vulnerabilities. These vulnerabilities include information leakage through older file versions that shouldn't be accessible, buggy code, and usage of deprecated functions. This challenge shows that such issues must be prevented as they can have serious security consequences. It also nicely shows that sometimes, multiple vulnerabilities must be combined to carry out an attack.

6.2 Required solution for full points

- The complete URL you used to successfully execute the attack and a brief description why this URL can be used to do the attack.
- The secret flag.

6.3 Hints

- Try to find the filename of the home page resource. You could try typical names such as *index.html*, but there's a nice tool that can help you (and which you already used in a previous challenge): *gobuster*. To search for common files and directories in a web application / on a web server that is reached at *https://www.site.com*, use it as follows in a terminal (option *-w* specifies the file with the directories/files that should be tried):
  gobuster dir -e -u https://www.site.com -w /usr/share/wordlists/dirb/common.txt
  Applying this to the Historia Animalum website should reveal the filename of the home page resource.
- At the bottom, the home page says «Made with vim», i.e., with the *vim* editor. Depending on the settings, *vim* may create backup files in the local directory when editing a file, and often the names of such backup files use the original filename with special characters or digits appended to the end. In case you know typical backup filenames used by *vim*, you can directly try to access a possibly existing backup file of the home page resource. If not, *gobuster* comes to the rescue once more! *gobuster* has an additional option *-d*, which, based on found resources, also tries to access corresponding backup files. So simply run the same command as above using the additional option *-d*.
- Once you have identified the filename of the backup file, inspect its source code, which reveals interesting information. Apparently, some PHP code was included in a previous version as text (instead of code), but it may be that this code is now included as actual code in the current version of the home page resource.
- Study the revealed information (source code) carefully. It appears that using suitable query parameters *wolve*, *user* and *pass* should provide you with interesting information, in this case the required flag to solve the challenge (see *echo* statement at the bottom of the code). But you don't know any username or a password, so another strategy is needed.
- The revealed code is buggy and insecure:
  - A deprecated version of *parse_str* is used. In this deprecated variant, *parse_str* gets a query string, e.g., *var1=foo&var2=bar*, as parameter, and based on this, one variable *$var1* with value *foo* is created and another variable *$var2* with value *bar* is created. This means that an attacker can, by using appropriate query parameters in the URL, create arbitrary variables with arbitrary values within the executing PHP program. This is highly security critical. Note: There is a newer version of *parse_str* available which is secure, but the deprecated version is still supported until PHP 7.2.x.
  - Beyond this, *parse_str* is used in a completely wrong way. It has no return value, so the variable *$string* won't get a value. As a result, all accesses in the form *$string[' .. ']* also won't return a value and any subsequent calls of *!empty($string[' .. ']*) return *false*. As a result of this, variables *$page*, *$user* and $pass won't be created.
  - The line *if ($page === '$_SERVER[REMOTE_ADDR]')* should probably make sure that requests should only be accepted from specific addresses. However, for this, one would have to use *$_SERVER['REMOTE_ADDR']*, i.e., the quotes are at the wrong place. In any case, this bug has the effect that if the variable *$page* has the value *$_SERVER[REMOTE_ADDR]*, the *if* condition is true and the block with the *echo* statement at the end is entered.
  - Looking at the code below *if ($page === '$_SERVER[REMOTE_ADDR]')*, one can see that the idea here is to use the values of the query parameters *user* and *pass*, and do some hashing based on them to come up with a result that is stored in variable *$secret*. However, as described above, *!empty($string['user'])* and *!empty($string['pass'])* will always be *false* and therefore, the three *if* blocks above the *echo* statement will never be entered and as a result of this, no variable *$secret* will be created.
- This should provide you with enough hints to do the attack: Try make sure that the *echo* statement is reached and that the secret flag is included in the web page.

Lab Points

In this lab, you can get **2 Lab Points**. To get them, you must achieve all 700 Hacking-Lab points that you can get from solving the challenges in this lab.